MVC - El patrón Modelo-Vista-Controlador

Modelo-vista-controlador (MVC) es el patrón de arquitectura de software más utilizado en la actualidad en desarrollo web (y también en muchas aplicaciones de escritorio). Este patrón propone separar la aplicación en tres componentes distintos: el modelo, la vista y el controlador:

- El modelo es el conjunto de todos los datos o información con la que trabaja la aplicación. Normalmente serán variables extraídas de una base de datos y el modelo gestiona los accesos a dicha información, tanto consultas como actualizaciones, implementando también los privilegios de acceso que se hayan descrito en las especificaciones de la aplicación (lógica de negocio). Normalmente el modelo no tiene conocimiento de las otras partes de la aplicación.
- La vista muestra al usuario el modelo (información y lógica de negocio) en un formato adecuado para interactuar (usualmente la interfaz de usuario). Es la intermediaria entre la aplicación y el usuario
- El controlador es el encargado de coordinar el funcionamiento de la aplicación. Responde a los eventos del usuario para lo que hace peticiones al modelo (para obtener o cambiar la información) y a la vista (para que muestre al usuario dicha información).

Este patrón de arquitectura de software se basa en las ideas de reutilización de código y la separación de conceptos, características que buscan facilitar la tarea de desarrollo de aplicaciones y su posterior mantenimiento.

Una aplicación sin MVC

Si una aplicación no utiliza este modelo, cuando una función modifica los datos debe además reflejar dicha modificación en la página para que la vea el usuario.

Por ejemplo: vamos a hacer una aplicación para gestionar un almacén. Entre otras muchas cosas tendrá una función (que podemos llamar **addProduct**) que se encargue de añadir un nuevo producto al almacén. Dicha función deberá realizar:

- añadir el nuevo producto al almacén (por ejemplo, añadiéndolo a un array de productos)
- pintar en la página ese nuevo producto (por ejemplo, añadiendo una nueva línea a una tabla donde se muestran los productos)

```
// La función que se ejecuta cuando el usuario envía el
// formulario para añadir un producto debería hacer:

// Coge los datos del formlario
const name = document.getElementById('product-form-name').value
const price = document.getElementById('product-form-name').price
...
// Valida cada dato
if (!name || name.length < 5 || ...)
...
// Añade el producto a la BBDD
const prod = await addProductToDatabase(payload)
let newProd = new Product(prod.id, prod.name, prod.price, prod.units)
this.products.push(newProd)
...
// Pinta en la página el nuevo producto
const DOMproduct = document.createElement('tr')
...
document.getElementById('products-table').apendChild(DOMproduct)
// Gestiona los posibles errores producidos en todo este proceso
...
```

Como vemos, se va a convertir en una función muy grande y que se encarga de muchas cosas distintas por lo que va a ser difícil mantener ese código. Además, toda la función es muy dependiente del HTML (en muchas partes se buscan elementos por su id).

Nuestro patrón MVC

En una aplicación muy sencilla puede no seguirse este modelo, pero en cuanto la misma se complica un poco es imprescindible programar siguiendo buenas prácticas ya que en otro caso el código se volverá rápidamente muy difícil de mantener.

Hay muchas formas de implementar este modelo. Si se trata de un proyecto con OOP se debe seguir el patrón MVC usando clases. Si sólo se utiliza programación estructurada será igual, pero en vez de clases y métodos habrá funciones.

Para organizar el código se crean 3 subcarpetas dentro de la carpeta src:

- **model**: aquí incluiremos las clases que constituyen el modelo de nuestra aplicación
- **view**: aquí crearemos un fichero JS que será el encargado de la GUI de nuestra aplicación, el único dependiente del HTML. Este fichero será una clase que representa toda la vista, aunque en aplicaciones mayores lo normal es tener clases para cada página, etc.
- **controller**: aquí estará el fichero JS que contendrá el controlador de la aplicación

Al hacerlo así, para cambiar la forma en que se muestra algo, se hace directamente en la vista modificando la función que se ocupa de ello.

La vista será una clase cuyas propiedades serán elementos de la página HTML a los que se accederá frecuentemente, para no tener que buscarlos cada vez y para que estén disponibles para el controlador. Contendrá métodos para renderizar los distintos elementos de la vista.

El controlador será una clase cuyas propiedades serán el modelo y la vista, de forma que pueda acceder a ambos elementos. Tendrá métodos para las distintas acciones que pueda hacer el usuario (y que se ejecutarán como

respuesta a dichas acciones, tal como se explica en el tema de eventos). Cada uno de esos métodos llamará a métodos del modelo (para obtener o cambiar la información necesaria) y posteriormente de la vista (para reflejar esos cambios en lo que ve el usuario).

El fichero principal de la aplicación instanciará un controlador y lo inicializará.

Por ejemplo, siguiendo con la aplicación para gestionar un almacén. El modelo constará de:

- La clase **Store** que es nuestro almacén de productos (con métodos para añadir o eliminar productos, etc).
- La clase **Product** que gestiona cada producto del almacén (con métodos para crear un nuevo producto, etc).

El fichero principal sería algo como:

main.js

```
const storeApp = new Controller()       // crea el controlador
storeApp.init()             // lo inicializa

// En desarrollo podemos añadir algunas líneas que luego quitaremos para
// imitar acciones del usuario y así ver el funcionamiento de la aplicación:
storeApp.addProductToStore({ name: 'Portátil Acer Travelmate E2100', price: 523.12 })
storeApp.changeProduct({ id: 1, price: 515.95 })
storeApp.deleteProduct(1)
```

model/store.class.js

```
export default class Store {
    constructor (id) {
        this.id=Number(id)
        this.products=[]
    }

    addProduct(payload) {
        // llama a la BBDD para que añada el producto
        const prod = await addProductToDatabase(payload)
        let newProd = new Product(prod.id, prod.name, prod.price, prod.units)
        this.products.push(newProd)
        return newProd
    }

    findProduct(id) {
        ...
    }
    ...
  }
```

model/product.class.js

```
export default class Product {
    constructor (id, name, price, units) {
```

```
        this.id = id
        this.name = name
        this.price = price
        this.units = units
    }
    ...
  }
```

view/index.js

```
export default class View {
    constructor {
        this.messageDiv = document.getElementById('messages')
        this.productForm = document.getElementById('product-form')
        this.productsList = document.getElementById('products-table')
    }

    init() {
        ...         // inicializa la vista, si es necesario
    }

    renderNewProduct(prod) {
        // código para añadir a la tabla el producto pasado añadiendo una nueva fila
        const DOMproduct = document.createElement('tr')
        ...
        this.productsList.apendChild(DOMproduct)
    }
    ...

    showMessage(type, message) {
        // código para mostrar mensajes al usuario y no tener que usar los alert
        const DOMmessage = document.createElement('div')
        ...
        this.messageDiv.apendChild(DOMmessage)
    }
  }
```

controller/index.js

```
export default class Controller {
    constructor() {
        this.store = new Store(1)       // crea el modelo, un Store con id 1
        this.view = new View()      // crea la vista
    }

    init() {
        this.view.init()            // inicializa la vista, si es necesario

        // Pone los escuchadores para poder interactuar con el usuario
        // Por ejemplo para enterarse de si el usuario envía el formulario:
```

```
        this.view.productForm.addEventListener('submit', (event) => {
            event.preventDefault()

            // Pide a la vista los datos del formulario
            const payload = this.view.getProductForm()
            // Y llama al método que se ocupa de añadir un producto
            this.addProductToStore(payload)
        })
    }

    addProductToStore(prod) {
        // haría las comprobaciones necesarias sobre los datos y luego
        try {
            // dice al modelo que añada el producto
            const newProd = this.store.addProduct(prod)
            // si lo ha hecho le dice a la vista que lo pinte
            this.view.renderNewProduct(newProd)
        } catch(err) {
            this.view.showErrorMessage('error', 'Error al añadir el producto')
        }
    }
    ...
}
```

Se puede obtener más información y ver un ejemplo más completo en https://www.natapuntes.es/patron-mvc-en-vanilla-javascript/